

DAM
Desarrollo de Aplicaciones Multiplataforma
2º Curso

AD
Acceso a Datos

UD 4
Modelo Objeto Relacional (ORM)

IES BALMIS
Dpto Informática
Curso 2019-2020
Versión 1 (03/2019)

UD4 – Modelo Objeto Relacional (ORM)

ÍNDICE

1. Mapeo Objeto-Relacional
2. JPA
3. Contacto con Hibernate
4. Nociones básicas de HQL
5. Actualizaciones de datos con Hibernate
6. Problemas frecuentes con Hibernate
7. Hibernate con dos tablas
8. Hibernate y relaciones “muchos a muchos”
9. Anotaciones e Hibernate

1. Mapeo Objeto-Relacional

Es enormemente frecuente emplear aplicaciones cuya lógica ha sido diseñada mediante Programación Orientada a Objetos, pero cuyos datos, sea porque existieran desde antes o sea por consideraciones de rendimiento y de facilidad de consulta, están almacenados en bases de datos relacionales. Eso supone que en ocasiones sea necesario convertir datos que internamente estaban en forma de objeto pero que externamente se van a representar como tablas.

Una célebre frase de la autora americana **Esther Dyson** dice: “Usar tablas para almacenar objetos es como conducir tu coche para volver a casa y luego desmontarlo para guardarlo en el garaje. Se puede volver a montar por la mañana, pero finalmente uno se acaba preguntando si ésta es la forma más eficiente de aparcar un coche”.

Frase original: “Using tables to store objects is like driving your car home and then disassembling it to put it into the garage. It can be assembled again in the morning, but one eventually asks whether this is the most efficient way to park a car”, citada en el libro “Object-Oriented Application Development Using the Caché Postrelational Database”, editado por Springer, 1999, página 23.

En el interior de un ordenador, las cosas ocurren mucho más deprisa que “en el mundo real”, por lo que no es realmente comparable el tiempo de convertir objetos a tablas y viceversa con el tiempo que una persona tardaría en desmontar y volver a montar un coche, pero esta frase da una idea de que quizá se esté realizando un trabajo evitable y relativamente pesado.

Se denomina **desfase objeto-relacional** (en inglés “Object-relational impedance mismatch”) a la serie de dificultades conceptuales y técnicas que aparecen al emplear un gestor de bases de datos relacional como medio de almacenamiento para una aplicación creada con un diseño orientado a objetos, debidas a que los objetos se deben “mapear” para convertirse en tablas del esquema relacional.

La idea de dicho “mapeo” es convertir los objetos en datos primitivos del gestor de bases de datos, que se puedan guardar en tablas de una base de datos relacional, así como volver a generar un objeto a partir de esos datos primitivos cuando se desee consultar la información.

2. JPA

La API de persistencia de Java (Java Persistence API, JPA) es una especificación de interfaz de programación de aplicaciones (API) para Java, que detalla cómo manipular datos relacionales en aplicaciones de Java SE y Java EE.

La especificación JPA 1.0 se lanzó en mayo de 2006, mientras que JPA 2.0 es de diciembre de 2009 y JPA 2.1 de abril de 2013.

Una **entidad de persistencia** es una clase Java ligera cuyo estado se puede volcar a una tabla en una base de datos relacional. Las instancias de una de estas entidades se corresponden con filas individuales de la tabla. Las entidades suelen tener relaciones con otras entidades, y estas relaciones se expresan a través de metadatos objeto/relacionales. Dichos metadatos se pueden especificar directamente en el archivo de clase usando "**anotaciones Java**" (etiquetas especiales precedidas por @), o bien en un archivo XML de descripción, que se distribuiría con la aplicación.

JPQL (abreviatura de "Java Persistence Query Language") es un lenguaje de consulta orientado a objetos, independiente de la plataforma, definido como parte de la especificación JPA. Las consultas se realizan a entidades almacenadas en una base de datos relacional. Estas consultas se parecen en su sintaxis a las consultas SQL, pero trabajan con objetos de entidad en lugar de hacerlo directamente con las tablas de base de datos.

Existen distintas tecnologías relacionadas con JPA, como por ejemplo:

- **Enterprise JavaBeans (EJB):** Es una de las interfaces de programación (API) que forma parte de Java EE (Java Enterprise Edition). Incluye una serie de objetos que son útiles en la programación en el lado del servidor, y que pueden facilitar tareas como el proceso de transacciones, la gestión de la concurrencia, la invocación asíncrona de métodos, la planificación de tareas, servicios de directorio, seguridad, etc. La especificación EJB 3.0 (a su vez parte de la plataforma Java EE 5) incluye soporte de persistencia.
- **Java Data Objects API:** El API de persistencia de Java especifica la forma de implementar persistencia para sistemas gestores de bases de datos relacionales (aunque algunos proveedores dan soporte a otros modelos de bases de datos)
- **Hibernate ORM** (o simplemente Hibernate) es un framework de código abierto para mapeo objeto-relacional en Java. Las versiones 3.2 y posteriores implementan el API de persistencia de Java. Será la herramienta en la que nos centremos en este tema.

3. Contacto con Hibernate

La idea básica de Hibernate es que existirán **ficheros XML de configuración** que indiquen cómo se deben convertir los objetos a tablas y viceversa. Estos ficheros se pueden crear a mano, pero NetBeans incluye **asistentes** para hacer el proceso más sencillo.

En este primer contacto, vamos a usar NetBeans para crear un proyecto sencillo que conecte a una base de datos que tenga una única tabla.

3.1. Creación de la BD

Crearemos en MySQL una BD denominada **bibliotecah** y en ella una tabla llamada libros.

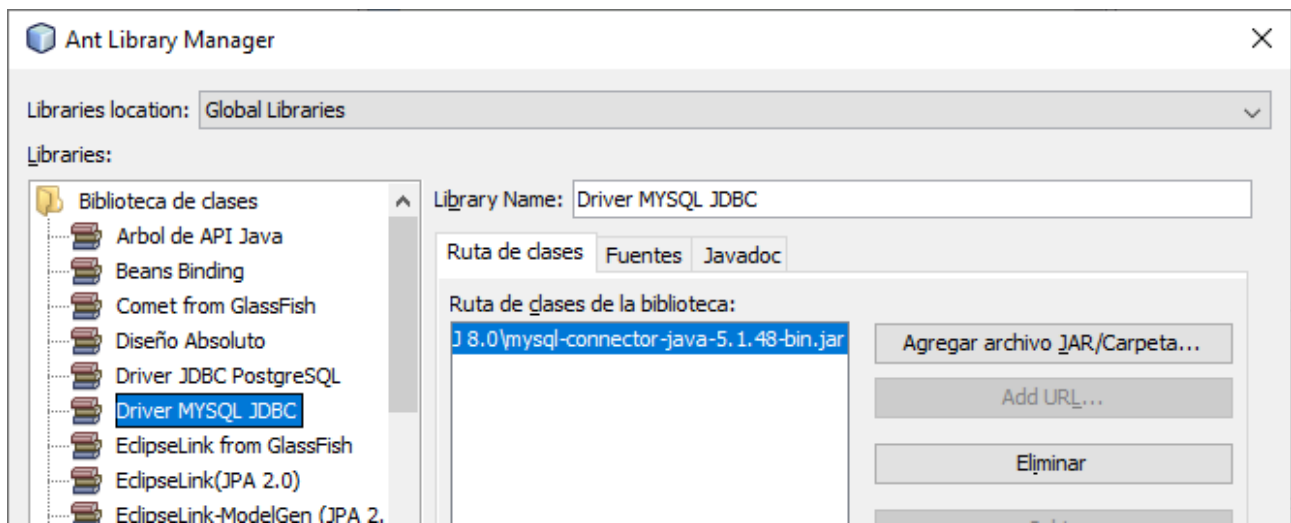
```
CREATE TABLE libros (  
    id          INTEGER PRIMARY KEY,  
    titulo      VARCHAR(60),  
    autor       VARCHAR(60)  
);
```

Y luego introducimos una serie de datos de ejemplo.

```
INSERT INTO libros ( id, titulo, autor ) VALUES  
(1, 'Macbeth', 'William Shakespeare'),  
(2, 'La Celestina (Tragicomedia de Calisto y Melibea)', 'Fernando de Rojas'),  
(3, 'El Lazarillo de Tormes', 'Anónimo'),  
(4, '20.000 Leguas de Viaje Submarino', 'Julio Verne'),  
(5, 'Alicia en el País de las Maravillas', 'Lewis Carroll'),  
(6, 'Cien Años de Soledad', 'Gabriel García Márquez'),  
(7, 'La tempestad', 'William Shakespeare');
```

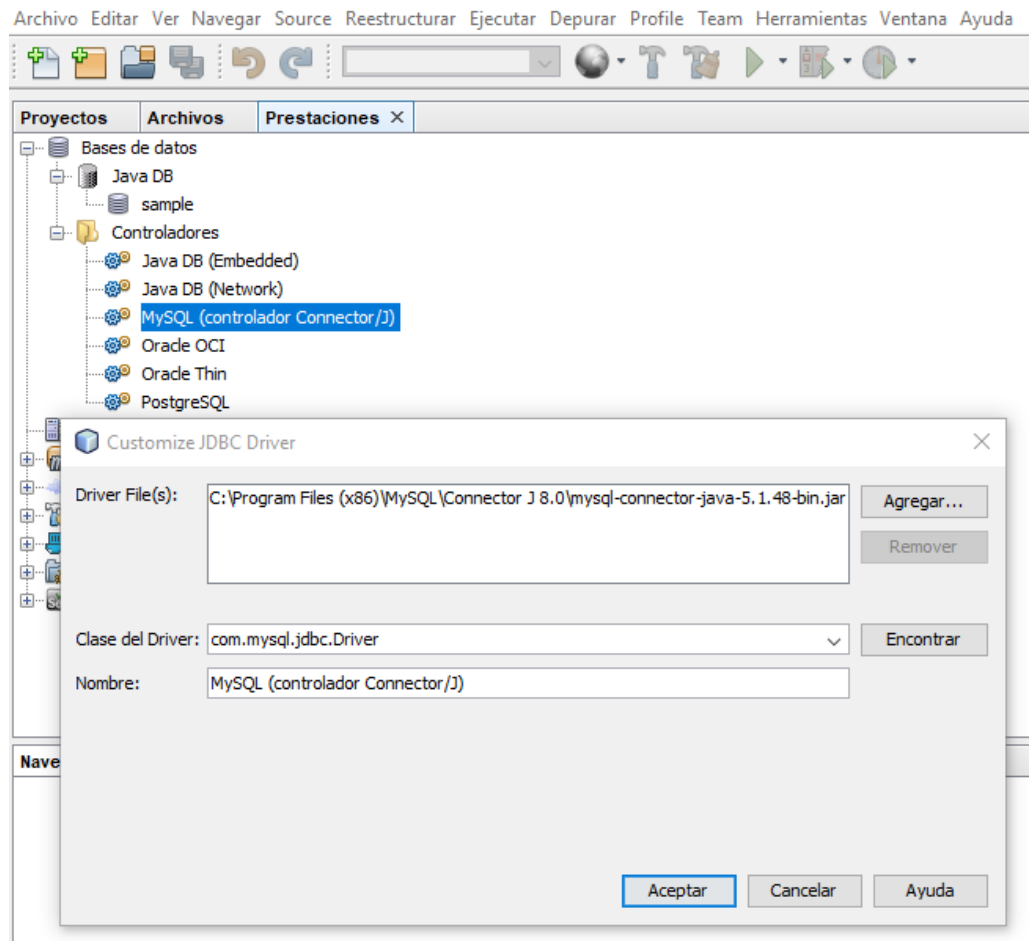
3.2 Conectar con la BD desde Netbeans

En Netbeans accederemos a "Herramientas → Libraries" y actualizaremos el driver JDBC de MYSQL.

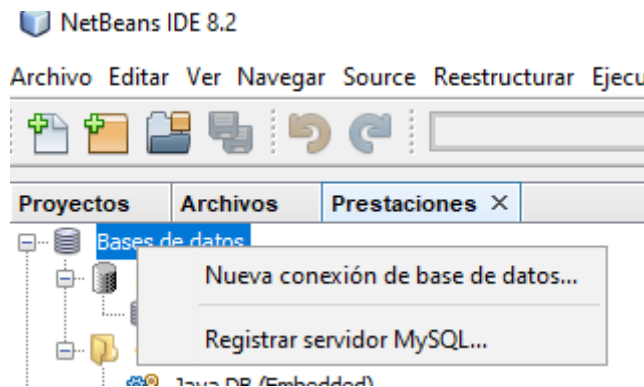


Luego en la pestaña "**Prestaciones**" de la barra izquierda.

Comprobaremos que tenemos actualizado el driver JDBC de MySQL. Si no fuera así, lo actualizaríamos de nuevo en esta pantalla también.



Ahora añadiremos una "**Nueva conexión de Base de Datos**" de tipo "MySQL (controlador Connector/J)" pulsando con el botón derecho sobre "Base de Datos".



En la siguiente pantalla completaremos los datos de conexión, añadiendo a la URL

&autoReconnect=true&useSSL=false

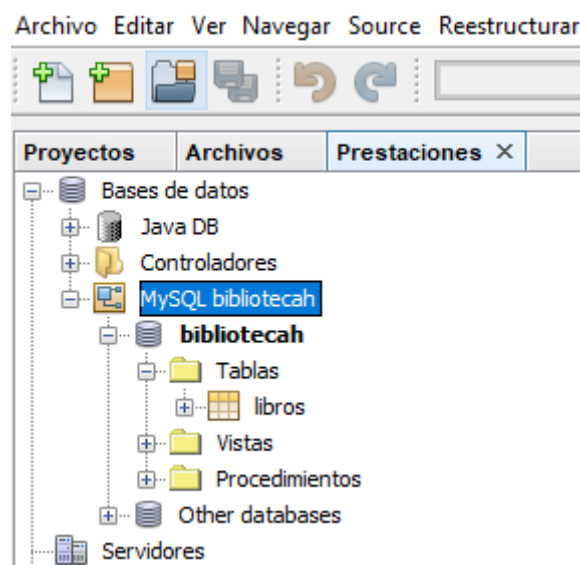
que le indicará que reconecte automáticamente si se pierde la conexión y que no utilice SSL.

Antes de pulsar en "**Siguiente**", podemos pulsar en "**Probar conexión**" y comprobar que funciona correctamente.

La URL de conexión quedará:

`jdbc:mysql://localhost:3306/bibliotecah?
zeroDateTimeBehavior=convertToNull&autoReconnect=true&useSSL=false`

En la última pantalla indicaremos el nombre de la conexión. En nuestro caso "**MySQL bibliotecah**".



3.3. Crear el proyecto

Crearemos un proyecto vacío de tipo "**Java Application**" con el nombre **Hibernate1** y un nuevo "**Java Package**" denominado **bibliotecah**.

3.4. Crear fichero de configuración

El primer paso será indicar que se desea crear un "nuevo fichero" ("New File"):

Y seleccionaremos la categoría "**Hibern (Hibernar)**". El primer tipo de fichero que encontraremos en esa categoría será el "**Hibernate Configuration Wizard** (asistente de configuración de Hibernate)".

Solo habrá que elegir en qué carpeta guardarlo (se nos propone la carpeta de fuentes del proyecto, "src") y le indicaremos nuestro package "**src/bibliotecah**".

Después deberemos escoger la conexión de la base de datos que debemos usar, donde seleccionaremos la que hemos creado con el nombre "**MySQL bibliotecah**".

Al final se creará el archivo **hibernate.cfg.xml** con la configuración básica.

Aunque también se puede hacer de forma manual desde el editor de textos "**Source**", desde el editor visual en la pestaña "**Design / Diseño**" podremos añadir otras propiedades.

Al grabar el archivo **hibernate.cfg.xml** podremos ver en **Source** que quedaría:

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-configuration>
  <session-factory>
    <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
    <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
    <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/bibliotecah?
zeroDateTimeBehavior=convertToNull&autoReconnect=true&useSSL=false</property>
    <property name="hibernate.connection.username">root</property>
    <property name="hibernate.connection.password">1234</property>
    <property name="hibernate.show_sql">true</property>
    <property name="hibernate.query.factory_class">
      org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</property>
    </session-factory>
  </hibernate-configuration>
```


3.5. POJOs y fichero de ingeniería inversa

La idea detrás de Hibernate es que existirá una clase Java equivalente a cada tabla de la base de datos, de modo que Hibernate se encargará de guardar los datos o de recuperarlos convirtiendo de un formato a otro cuando sea necesario.

Por tanto, para nuestra tabla de “libros” tendremos una clase “**Libro**”, que, en principio, tendrá atributos similares a los campos de la tabla (**id**, **autor**, **título**).

Estas clases que Hibernate sea capaz de llevar a la base de datos deben ser “sencillas”, que no dependan de ningún framework concreto, no extiendan ni implementen nada especial. Es lo que se conoce como **POJOs (Plain Old Java Objects)**. En general, para que sean utilizables a nivel de persistencia, también deberán tener un constructor sin parámetros, métodos “**get**” y “**set**” para todos sus atributos (realmente no para todos, sino para aquellos que se desee almacenar en la base de datos, que podrían no ser la totalidad). En algunas ocasiones también es necesario crear métodos “**equals()**” y “**hashCode()**”.

La ventaja de haber creado en primer lugar la base de datos es que ahora se puede crear de forma semiautomática las clases sencillas que representen la lógica equivalente dentro de nuestra aplicación.

Dentro de los tipos de ficheros de Hibernate que se pueden crear desde NetBeans, veremos la opción de crear:

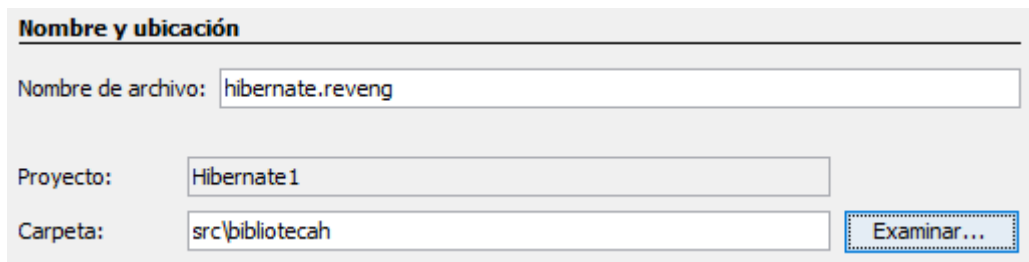
- los “**ficheros y mapeado**”
- y los objetos “**POJOs**” a partir de la base de datos.

El fichero de mapeado será el que indique, para cada atributo de una clase, cuál es la columna correspondiente en la tabla que le representa. Se podría crear a mano o de forma semiautomática, pero si lo intentamos ahora obtendremos un mensaje de aviso que nos dice que primero debemos crear un “**fichero de ingeniería inversa**”.

Para crear los ficheros de ingeniería inversa usaremos el correspondiente asistente, el “**Hibernate Reverse Engineering Wizard** (Asistente de ingeniería inversa de Hibernate)”.

Se nos propondrá un nombre de fichero (terminado en “**.reveng.xml**”, como abreviatura de “reverse engineering”) y una ruta, que podemos confirmar.

Si queremos cambiar la ruta, por ejemplo para dejar el fichero de configuración dentro de la misma carpeta de los fuente ("src"), pulsaríamos el botón "**Browse**" y escogeríamos la carpeta deseada ("src/bibliotecah").



Nombre y ubicación

Nombre de archivo:

Proyecto:

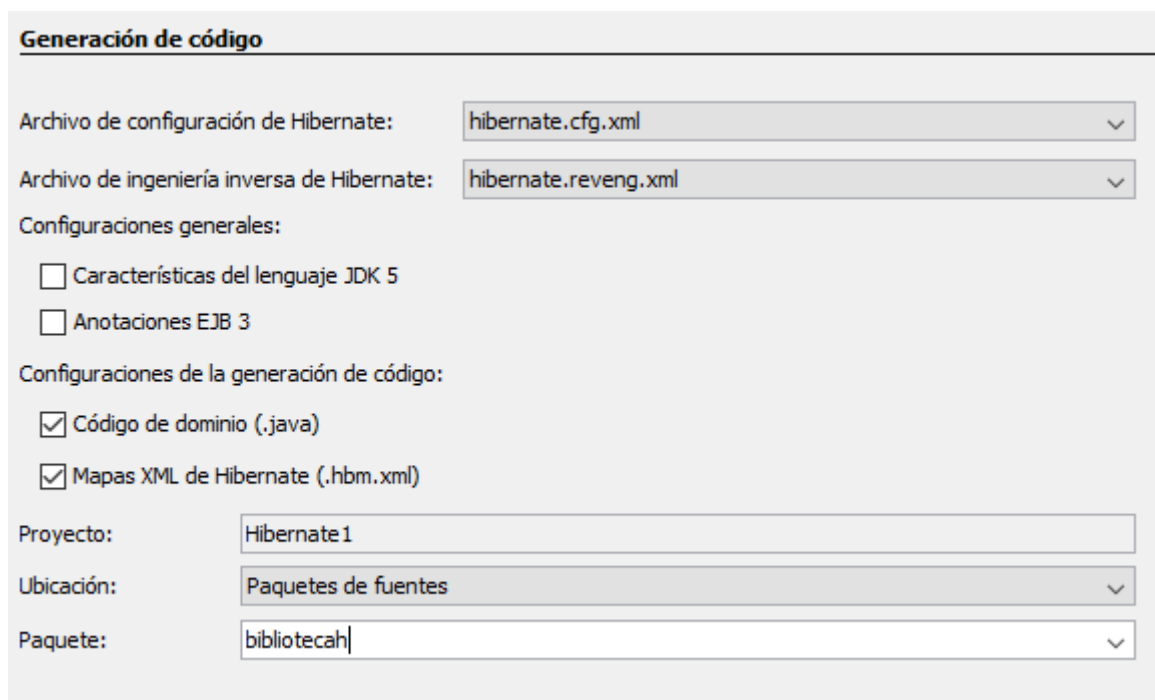
Carpeta:

Y luego deberemos elegir qué tablas queremos analizar (en nuestro caso solo hay una, **libros**).

Y obtendremos un fichero de configuración muy compacto.

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-reverse-engineering>
  <schema-selection match-catalog="bibliotecah"/>
  <table-filter match-name="libros"/>
</hibernate-reverse-engineering>
```

Pero ahora, con la ayuda de ese fichero, ya sí podemos pedir que nos cree los POJOs correspondientes y el fichero de mapeado. El asistente es "**Archivos de mapas de Hibernate y POJOs de la Base de Datos**"



Generación de código

Archivo de configuración de Hibernate:

Archivo de ingeniería inversa de Hibernate:

Configuraciones generales:

☐ Características del lenguaje JDK 5

☐ Anotaciones EJB 3

Configuraciones de la generación de código:

☒ Código de dominio (.java)

☒ Mapas XML de Hibernate (.hbm.xml)

Proyecto:

Ubicación:

Paquete:

Nos preguntará qué fichero de configuración de Hibernate y qué fichero de ingeniería inversa utilizar (los podrá detectar automáticamente).

Y si pulsamos "**Finish**", obtendremos una clase "**Libros**", con sus "**set**", sus "**get**" y tres constructores (uno vacío, otro sólo con el "id" y otro con todos sus atributos).

```
package hibernate1;

import java.io.Serializable;

public class Libros implements Serializable {

    private int id;
    private String titulo;
    private String autor;

    // Constructores
    public Libros() {
    }
    public Libros(int id) {
        this.id = id;
    }
    public Libros(int id, String titulo, String autor) {
        this.id = id;
        this.titulo = titulo;
        this.autor = autor;
    }

    // getters y setters
    public int getId() {
        return this.id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getTitulo() {
        return this.titulo;
    }
    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }
    public String getAutor() {
        return this.autor;
    }
    public void setAutor(String autor) {
        this.autor = autor;
    }
}
```

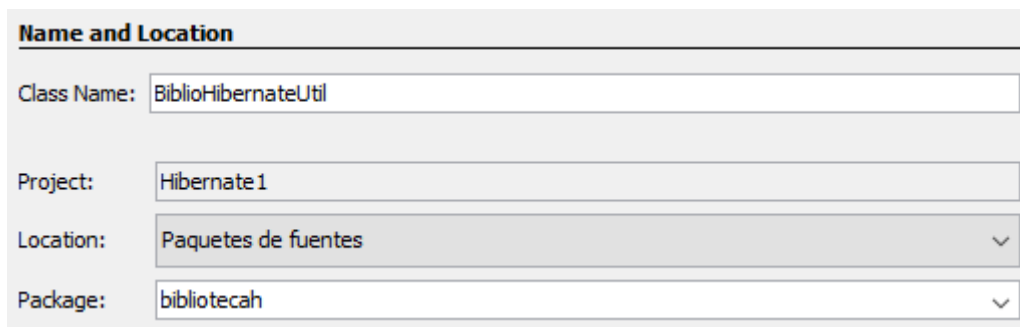
También nos crea el archivo de mapeado **Libros.hbm.xml** donde se realiza la asignación de cada propiedad de la clase Libros a qué campo se asigna.

```
<?xml version="1.0" encoding="UTF-8"?>
<hibernate-mapping>
  <class name="bibliotecah.Libros"
        table="libros"
        catalog="bibliotecah"
        optimistic-lock="version">
    <id name="id" type="int">
      <column name="id" />
      <generator class="assigned" />
    </id>
    <property name="titulo" type="string">
      <column name="titulo" length="60" />
    </property>
    <property name="autor" type="string">
      <column name="autor" length="60" />
    </property>
  </class>
</hibernate-mapping>
```

Un objeto fundamental para acceder a Hibernate son las sesiones. Debemos crear una sesión, emplearla para leer y guardar datos, y finalmente deberemos cerrarla.

Para que el manejo de sesiones sea un poco más cómodo, se recomienda emplear un asistente adicional denominado **HibernateUtil.java**.

Crearemos el archivo **BiblioHibernateUtil.java** y se nos recordará que es recomendable que sea parte de un package, así que podríamos añadirla al package "**bibliotecah**".



Name and Location

Class Name:

Project:

Location:

Package:

Y obtendremos un fichero como éste:

```
package bibliotecah;

import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;

public class BiblioHibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            sessionFactory =
                new AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

En este fichero realizaremos algunos cambios para no utilizar métodos deprecated. Además, añadimos al método configure como parámetro el fichero de nuestra configuración "**bibliotecah/hibernate.cfg.xml**".

```
BiblioHibernateUtil.java

package bibliotecah;

import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;
import org.hibernate.service.ServiceRegistry;

public class BiblioHibernateUtil {

    private static SessionFactory sessionFactory;

    public static SessionFactory getSessionFactory() {

        Configuration configuration=
            new Configuration().configure("bibliotecah/hibernate.cfg.xml");

        StandardServiceRegistryBuilder serviceRegistryBuilder =
            new StandardServiceRegistryBuilder();

        serviceRegistryBuilder.applySettings(configuration.getProperties());

        ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();
        sessionFactory = configuration.buildSessionFactory(serviceRegistry);

        return sessionFactory;
    }

    public static void closeSessionFactory() {
        sessionFactory.close();
        StandardServiceRegistryBuilder
            .destroy(sessionFactory.getSessionFactoryOptions().getServiceRegistry());
    }
}
```

3.6 Obtener datos

Para obtener datos, será frecuente que partamos de una consulta que permita filtrar qué datos exactos deseamos obtener. Para ello, Hibernate permite emplear un lenguaje de consulta similar a SQL, llamado a **HQL**.

Veremos algunos detalles más adelante, pero una de las primeras diferencias de sintaxis que debemos tener en cuenta es que si queremos obtener todos los datos de una tabla, no haremos "**select * from tabla**", sino simplemente "**from tabla**".

```
import bibliotecah.BiblioHibernateUtil;
import bibliotecah.Libros;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;

public class Hibernate1 {

    private static Session sesion;

    // *****
    // verLibros -> Mostrar datos de todos los libros
    // *****
    public static void verLibros() {

        System.out.println ("Mostrando todos los libros:");

        Query consulta = sesion.createQuery("from Libros");

        List resultados = consulta.list();
        for(Object resultado : resultados) {
            Libros libro = (Libros) resultado; // Cast
            System.out.println ( libro.getId() + ": " + libro.getTitulo()
                                + ", de " + libro.getAutor());
        }
    }

    // *****
    // MAIN
    // *****
    public static void main(String[] args) {
        // Eliminar mensajes de control de Hibernate
        org.jboss.logging.Logger logger =
            org.jboss.logging.Logger.getLogger("org.hibernate");
        java.util.logging.Logger.getLogger("org.hibernate")
            .setLevel(java.util.logging.Level.SEVERE);

        // Hibernate - Abrir sesión
        sesion = BiblioHibernateUtil.getSessionFactory().openSession();

        verLibros();

        // Hibernate - Cerrar sesión
        sesion.close();
        BiblioHibernateUtil.closeSessionFactory();
    }
}
```

Como se ve en el método **verLibros()** de este ejemplo, habrá que crear un objeto de tipo "**Query**" en la sesión, indicarle la consulta deseada y los resultados irán a parar a una lista, que se podrá recorrer como se desee.

También es importante el detalle de que en los nombre de clases y de atributos se debe distinguir entre **mayúsculas** y **minúsculas**. Por eso, la entidad (o Clase) en la que hay que buscar datos es "**Libros**", que es el nombre de nuestra **clase original**.

Si se quiere obtener solo un campo, se puede usar una sentencia SELECT idéntica a las de SQL.

```
Query consulta = sesion.createQuery("select titulo from Libros");
```

3.7 Resumen de HIBERNATE

De forma general, los pasos a seguir serán

- 1) **Crear conexión a Base de Datos en pestaña "PRESTACIONES"**
 - Pulsar en Conectar antes de continuar
- 2) **Asistente** → **Asistente de configuración de Hibernate**
 - hibernate.cfg.xml
- 3) **Asistente** → **Asistente de ingeniería inversa de Hibernate**
 - hibernate.revenge.xml
- 4) **Asistente** → **Archivos de mapas de Hibernate y POJOs de la Base de Datos**
 - Clase1.java + Clase1.hbm.xml
 - Clase2.java + Clase2.hbm.xml
 - ...
- 5) **Asistente** → **HibernateUtil.java**
 - (también se puede hacer de forma manual copiando la plantilla)
- 6) **Pruebas de funcionamiento**
 - Ejecutar la aplicación aunque todavía no haga nada (main vacío)
 - hibernate.cfg.xml (Botón derecho)
 - Abrir "Ejecutar consulta HSQL" y probar
 - from Clase1

3.8 Cambiar configuración de HIBERNATE

Mostrar consultas HSQL de Hibernate

Para mostrar las consultas HSQL que ejecuta Hibernate deberemos editar el archivo hibernate.cfg.xml.

Si lo hacemos desde el asistente (pestaña Diseño):

- **Optional properties** → Configuration properties:
Add → Hihernate.show_sql = true (para logging de sentencias SQL)

Si lo hacemos desde el editor (Source):

```
...  
<property name="hibernate.show_sql">true</property>  
...
```

Mostrar información de LOG de Hibernate

Para que se vean mensajes los mensaje del LOG de Hibernate bastará con cambiar el nivel de LOG en el main:

```
java.util.logging.Logger.getLogger("org.hibernate")  
    .setLevel(java.util.logging.Level.INFO);
```


4. Nociones básicas de HQL

Una vez hayamos realizado todo el proceso de configuración y mapeo de Hibernate indicado en el resumen anterior, podemos probar a realizar consultas HQL desde la consola gráfica. Para ello:

- Botón derecho sobre **hibernate.cfg.xml**
- Ejecutar la consulta HSQL

Vamos a ver algunas de las posibilidades de Hibernate Query Language (HQL), todavía orientadas a una única tabla.

Como ya hemos visto, es posible obtener todos los datos de un objeto con una sintaxis similar a la de SQL, pero omitiendo el "**SELECT ***" e indicando solamente **FROM** y el nombre de la tabla.

```
from Libros
```

Las **mayúsculas y minúsculas** se deben respetar en los nombres de las clases, pero las órdenes de HQL se pueden escribir tanto en mayúsculas como en minúsculas, de modo que también sería válido.

```
FROM Libros
```

Será habitual preferir obtener los datos ordenados, empleando la cláusula **ORDER BY**, al igual que en SQL.

```
FROM Libros ORDER BY autor
```

Y se puede añadir condiciones con **WHERE**. Así, un dato exacto se miraría con **"=**".

```
FROM Libros WHERE id=2
```

Si el dato a buscar es una cadena de texto, dado que las cadenas en Java se delimitan con comillas dobles, la encerraremos entre comillas simples en la consulta de HQL.

```
FROM Libros WHERE autor = 'Fernando de Rojas'
```

Se puede hacer búsquedas parciales con LIKE:

```
FROM Libros WHERE autor LIKE 'W%' ORDER BY titulo
```

Si se desea obtener sólo un campo, se usará la misma sintaxis que en SQL, como ya hemos visto (pero hay que tener en cuenta que en ese caso ya no obtendremos un objeto, sino una cadena de texto, un número entero o lo que corresponda):

```
SELECT titulo FROM Libros
```

Es posible emplear alias:

```
SELECT l.titulo FROM Libros l
```

Tenemos ciertas funciones de manipulación de cadenas, como “lower”, “upper”, “trim”, “substring”, “length”:

```
SELECT upper(titulo) FROM Libros
```

También se pueden realizar las operaciones aritméticas básicas (+ - / *) y usar algunas funciones matemáticas, como SQRT (raíz cuadrada), ABS (valor absoluto) y MOD (resto), además de otras que pueda permitir la base de datos que haya por debajo, como SIN (seno) o TRUNC (truncar para eliminar decimales).

```
SELECT id+1 FROM Libros
```

(Hay que recordar que en ese caso, el dato obtenido sería un “int”, no un “String”, y sería imprescindible hacer la conversión de tipos correcta).

También se pueden emplear funciones de agregación: MAX, MIN, AVG, COUNT, como en:

```
SELECT MAX(id) FROM Libros
```

HQL Tutorial

<https://docs.jboss.org/hibernate/orm/3.5/reference/en/html/queryhql.html>

Obtener datos sin mapear en sus clases usando Hibernate

Si no tenemos un objeto de la clase asociada al mapeo de la tabla, obtener dos o tres datos es un poco más complejo que obtener todo el objeto o un único campo.

El resultado que se recibe será un **array de objetos**, que se podrá recorrer mediante índice y se deberá convertir al tipo de datos correspondiente.

```
public static void verLibrosSELECT() {
    System.out.println("=====");
    System.out.println("Ver libros con GROUP BY");
    System.out.println("=====");

    Query consulta = sesion.createQuery("SELECT id, autor FROM Libros");

    List<Object[]> resultados = consulta.list();
    for( Object[] dato : resultados) {
        System.out.println ( (Integer) dato[0] );
        System.out.println ( (String) dato[1] );
    }
}
```

Teniendo esa precaución, se puede obtener también datos agrupados con GROUP BY e incluso imponer condiciones a los grupos con HAVING:

```
public static void verLibrosGROUPBY() {
    System.out.println("=====");
    System.out.println("Ver libros con GROUP BY");
    System.out.println("=====");

    Query consulta =
        sesion.createQuery("SELECT autor, count(*) FROM Libros GROUP BY autor");

    List<Object[]> resultados = consulta.list();
    for( Object[] dato : resultados) {
        System.out.println ( (String) dato[0] );
        System.out.println ( (Long) dato[1] );
    }
}
```

5. Actualizaciones de datos con Hibernate

5.1 Guardar o Insertar datos

Una vez utilizados los 4 asistentes, ya podremos comenzar nuestra aplicación.

Para guardar datos, deberemos disponer de una sesión y dentro de ella una transacción. A continuación prepararemos un objeto del tipo que nos interese (en nuestro caso, del tipo "Libros") y lo guardaremos con "**save**". Finalmente, deberemos hacer un "**commit**" de la transacción.

Una función de ejemplo, que pidiera al usuario los datos de un libro y los guardase, podría ser:

```
public static void addLibro() {
    int id = 0 ;
    String titulo, autor;
    Transaction trans = null;

    System.out.println("=====");
    System.out.println("Añadir un libro");
    System.out.println("=====");

    try {
        // Java - Leer datos
        Scanner teclado = new Scanner(System.in);

        System.out.print("Introduzca el código del libro: ");
        id = Integer.parseInt(teclado.nextLine());

        System.out.print("Introduzca el título: ");
        titulo = teclado.nextLine();

        System.out.print("Introduzca el autor: ");
        autor = teclado.nextLine();

        // Java - Cargar datos en un objeto Libro
        Libros libro = new Libros(id, titulo, autor);

        // Hibernate - Guardar datos del objeto directamente en la BD
        trans = sesion.beginTransaction();
        sesion.save(libro);
        trans.commit();
    } catch (NonUniqueObjectException e) {
        // Ya existe un libro con ese Id
        System.out.println("El Id "+id+" ya existe");
        trans.rollback();
    }
}
```

5.2 Modificar un dato existente

Para modificar un objeto (en nuestro caso un libro) se deberá recibir el objeto desde la base de datos, hacer los cambios que se desee y a continuación guardar usando ese mismo objeto:

```
public static void editLibro(int idBuscado, String newAutor) {
    System.out.println("=====");
    System.out.println("Modificar un libro");
    System.out.println("=====");

    // Hibernate - Obtener libro a modificar
    Query consulta = sesion.createQuery("FROM Libros WHERE id="+idBuscado);
    List resultados = consulta.list();
    Libros libroAModificar = (Libros) resultados.get(0);

    // Java - Modificar el libro
    libroAModificar.setAutor(newAutor);

    // Hibernate - Guardar datos del objeto directamente en la BD
    Transaction trans = sesion.beginTransaction();
    sesion.update(libroAModificar);
    trans.commit();
}
```

Para guardar los cambios se puede utilizar **".update"**, que sería la sintaxis más natural, o bien **".save"**, igual que cuando añadíamos un dato, siempre y cuando tengamos en cuenta que se debe estar modificando un objeto que se haya recibido de la base de datos.

5.3 Borrar un datos existente

Para borrar un dato, el proceso será casi el mismo: comenzar por recibir el objeto desde la base de datos, y luego utilizar "**sesion.delete(dato)**". En general, será deseable pedir confirmación, como en este ejemplo:

```
public static void borrarPorId(int IdBuscado) {
    System.out.println("=====");
    System.out.println("Eliminar un libro");
    System.out.println("=====");

    Scanner teclado = new Scanner(System.in);
    Query consulta = sesion.createQuery("FROM Libros WHERE id = " + IdBuscado);
    List<Libros> libros = consulta.list();
    if (libros.size() > 0) {
        System.out.println("¿Es este libro (S/N)? " + libros.get(0).getTitulo());
        String opcion = teclado.nextLine().toUpperCase();
        if (opcion.equals("S")) {

            Transaction trans = sesion.beginTransaction();
            sesion.delete(libros.get(0));
            trans.commit();

            System.out.println("Libro borrado");
        }
    } else {
        System.out.println("No existe un libro con ese código");
    }
}
```

5.4 Modificaciones en lote

Si se desea modificar muchos objetos, resultaría muy lento recibir cada uno de ellos, hacer los cambios pertinentes y guardar el objeto modificado. Es preferible usar para ello HQL, cuya sintaxis en general será la misma que la de SQL (con las mismas consideraciones que ya se han visto antes: que se debe usar los nombres de objeto y no de tabla, y que alguna función puede no estar disponible para todos los gestores de bases de datos).

La forma de lanzar una consulta de modificación será añadir **".executeUpdate()"** a un consulta creada con **".createQuery"**. Por ejemplo, si para los libros tuviéramos un atributo **"ubicación"** (la estantería / sección en la que está almacenado un libro) y quisiéramos convertir a mayúsculas todas las ubicaciones, se podría hacer con:

```
int datosModificados = sesion.createQuery("UPDATE Libros SET "
    + "ubicacion = UPPER(ubicacion)").executeUpdate();

System.out.println("Cantidad de libros actualizados: " + datosModificados);
```

En un caso más general, es esperable que sea necesario añadir una cláusula WHERE para que la modificación se realice sólo sobre algunos de los datos, no sobre todos ellos.

Además, en la condición de WHERE será preferible utilizar parámetros, como ya vimos en los “CallableStatements”, lo que tendrá ventajas como dificultar los ataques de inyección de SQL. Así, una versión más avanzada de la consulta anterior podría ser:

```
int datosModificados = sesion.createQuery("update Libros " +
    "set ubicacion = :nueva " +
    "where ubicacion = :antigua" )
    .setParameter("nueva", nuevaUbicacion )
    .setParameter("antigua", antiguaUbicacion )
    .executeUpdate();
```

Aun así, hay que tener presente que estas actualizaciones masivas pueden dar lugar a inconsistencias, porque los cambios no se reflejen inmediatamente en los objetos a los que accede el programa. Citando textualmente la documentación oficial de Hibernate:

Caution should be used when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a transaction in a new persistence context or before fetching or accessing entities whose state might be affected by such operations.

6. Problemas frecuentes con Hibernate

6.1. No consigo que funcione

Comprueba que has dado correctamente todos los pasos. Es habitual que sea una mala conexión a la base de datos, por elegir un driver incorrecto, o por no seleccionar la base de datos real sino la prefijada, o por no detallar la contraseña de acceso.

Compara tus ficheros de configuración con los de ejemplo.

6.2. El programa se queda activo y no se cierra

Debería bastar con cerrar la "SessionFactory" al final de Main:

```
session.close();  
BiblioHibernateUtil.closeSessionFactory();
```

6.3. Me molestan los mensajes de depuración de Hibernate.

Es interesante tener los mensajes "INFO" al menos hasta que el programa se comporte correctamente.

Una vez estemos en producción, la siguiente secuencia los elimina:

```
org.jboss.logging.Logger logger =  
    org.jboss.logging.Logger.getLogger("org.hibernate");  
java.util.logging.Logger.getLogger("org.hibernate")  
    .setLevel(java.util.logging.Level.SEVERE);
```